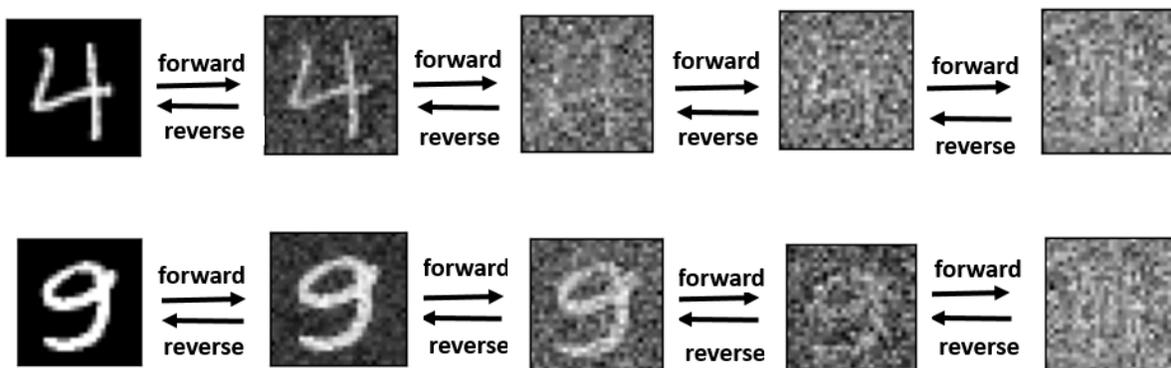


Implementing Denoising Diffusion Probabilistic Model

Sara Farahani, Milad Gholamrezanejad



Diffusion models are a type of generative models that draw inspiration from the principles of non-equilibrium thermodynamics. This process unfolds in two primary steps: the **forward process** and the **reverse process**, both operating as Markov chains.

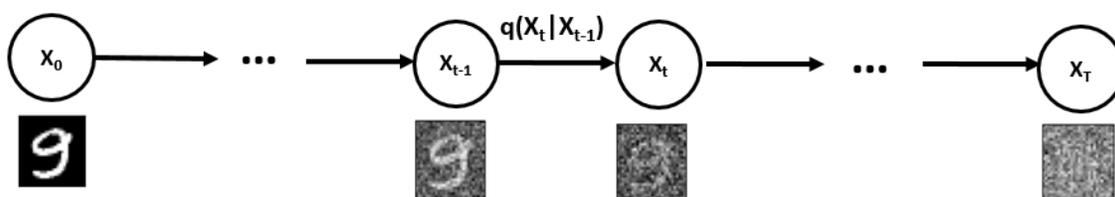
In this journey, we get familiar with these steps, implement them, train the model, and finally, generate new samples using a trained model. You can use pytorch or tensorflow libraries for the implementation.

1. Dataset

In this project, you will use two datasets: the [MNIST](#) dataset, which consists of handwritten digits from 0 to 9, and the Persian digits and letters dataset, which includes Persian digits from 0 to 9 and 32 Persian letters.

2. Forward Process

The forward step starts from a data point sampled from a real data distribution $x_0 \sim q(x)$. During this process, we add Gaussian noise to the sample in T timesteps, starting from the input data point and ending with pure Gaussian noise distribution. This Markov process is as follows:



$$q(x_{1:T}|x_0) = \prod_{t=1}^T q(x_t|x_{t-1}), \quad q(x_t|x_{t-1}) = N(x_t|\sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

Where:

- x_0 is the initial input image, x_t is the noisy image at timestep t , x_{t-1} is the noisy image at timestep $t-1$, which is less noisy compared to x_t .
- N is the Normal distribution with a mean of $\sqrt{1-\beta_t}x_{t-1}$ and a covariance matrix of $\beta_t I$.
- β_t is called "noise schedule" affecting the value of noise in each sample.

In the above formula of $q(x_t | x_{t-1})$, the noise is added sequentially, however, we can directly compute the noisy image at any arbitrary timestep t only from x_0 . It is possible by [reparameterization trick](#), which results in the following formula:

$$q(x_t | x_0) = N(x_t | \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t) I)$$

Where:

- N is the Normal distribution with $\bar{\alpha}_t$ mean and $(1 - \bar{\alpha}_t) I$ covariance matrix.
- $\alpha_t = 1 - \beta_t$
- $\bar{\alpha}_t = \prod_{s=1}^t \alpha_s$

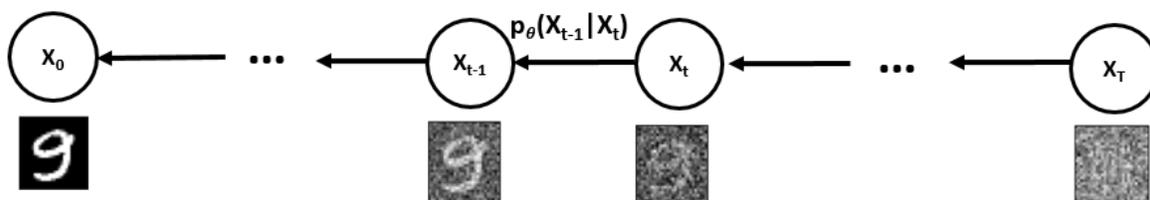
To implement the forward process, define the following variables, and complete the following functions in the `diffusion_model.py` file. Note that in this implementation we fix β_t . Therefore, the forward process has no learnable parameters.

```
class DiffusionModel(nn.Module):  
  
    def __init__(self, backward_process_model, beta_start, beta_end,  
                 timesteps=1000, device="cuda"):  
  
        self.backward_process_model = backward_process_model  
  
        self.betas = get_linear_beta_schedule(beta_start, beta_end, timesteps)  
  
        # TODO: Define alphas variable based on q(x_t|x_0) formula  
        # TODO: Define alpha_bars variable based on q(x_t|x_0) formula  
  
    def get_linear_beta_schedule(beta_start, beta_end, timesteps):  
  
        # TODO: Compute beta values by dividing the range from beta_start to  
        # beta_end into timesteps linearly  
  
        # TODO: Return betas
```

```
def add_noise(x_0, timestep):
    # TODO: Define noise as a random Gaussian noise with similar shape
    to x_0
    # TODO: Compute x_t by adding noise to x_0 based on q(x_t|x_0) formula
    # TODO: return x_t, noise
```

3. Reverse Process

The reverse process is defined as a Markov chain with a learned Gaussian transition model that starts with the last noisy image from the forward process $p(x_T) = N(x_T|0, I)$, which is pure Gaussian noise. The goal of this process is to denoise the samples in the backward direction.



$$p_{\theta}(x_{0:T}) = p(x_T) \prod_{t=1}^T p_{\theta}(x_{t-1}|x_t), \quad p_{\theta}(x_{t-1}|x_t) = N(x_{t-1}|\mu_{\theta}(x_t, t), \Sigma_{\theta}(x_t, t))$$

Where:

- x_0 is the initial input image, x_t is the noisy image at timestep t , x_{t-1} is the noisy image at timestep $t-1$, which is less noisy compared to x_t .
- N is the Normal distribution with $\mu_{\theta}(x_t, t)$ mean and $\Sigma_{\theta}(x_t, t)$ covariance matrix.

We need to use a neural network parameterized by θ to learn the parameters such that $p_{\theta}(x_0)$ is as close as possible to $q(x_0)$. In this project, we set $\Sigma_{\theta}(x_t, t) = \sigma_t^2 I$ and the only parameters that the model learns are those of $\mu_{\theta}(x_t, t)$. To implement the reverse process, you need to make changes in [UNET.py](#).

3.1. Implement Timestep Embedding

Before Implementing a neural network, consider that the parameters of the model are shared across time. As a result, we need to tell the network what time step we are in. To implement this, you need to encode the timesteps using positional embeddings. For this project, you may use sinusoidal positional embedding.

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$
$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

Where:

- pos is the position of the token in sequence (timesteps)
- i is the index of embedding dimension
- d is the embedding size

To implement this, complete the `get_sinusoidal_timesteps_embedding` function.

```
def get_sinusoidal_timesteps_embedding(timesteps, embedding_dim):  
    # TODO: Compute timesteps_embeddings using above formula  
    # TODO: Return timesteps_embeddings
```

3.2. Implement U-Net-based Model

Now, you need to implement a custom [U-Net](#)-based model as a neural network for $\mu_{\theta}(x_t, t)$. To do this, make changes in the `UNet` class. Remember that you need to use timesteps embeddings in the blocks of your neural network.

3.3. Implement Loss function

Now, we need a loss function to compute the difference between the added noise in the forward step and the result of the reverse process.

$$L_{simple}(\theta) = E_{t, x_0, \epsilon_t} [\|\epsilon_t - \epsilon_{\theta}(\sqrt{\alpha_t}x_0 + \sqrt{1 - \alpha_t}\epsilon_t, t)\|^2]$$

where :

- ϵ_t is the added noise in the forward process
- ϵ_θ is the predicted noise in the reverse process
- $||\cdot||$ denotes the Euclidean distance

```
def get_loss(noise, noise_pred):  
    # TODO: Compute l2-norm between noise and noise_pred
```

4. Train the model

Now, let's bring everything together and train the model. Train the model on each of the given datasets: the MNIST dataset and the Persian digits and letters dataset. To do this, complete the `forward` function in the `DiffusionModel` class. Then, you can run the `diffusion_model.py` file to train the model.

```
def forward(x):  
    # TODO: Define variable t by randomly selecting a timestep for each  
    image in a batch of images of x  
  
    # TODO: Use add_noise function to do the forward process by adding  
    noise to x  
  
    # TODO: Use reverse_process_model to do the reverse process by  
    predicting the noise of the forward step's results  
  
    # TODO: Return noisy x, added noise, predicted noise
```

5. Evaluate & Sample

For this step, you need to complete the following functions in the `diffusion_model.py` file. First, to evaluate your trained model, complete the `evaluate` function.

```
def evaluate(diffusion_model, test_loader, device="cuda"):  
    # TODO: Compute the accuracy of your trained model on the test data
```

Now, it's time to generate new samples using your trained model. To do this, complete the `sample` function.

```
@torch.no_grad()
def sample():
    # TODO: Use trained model to generate new samples
```

Presentation

For presentation, you should be able to:

- Provide a full explanation of all parts of the code
- Show the plots, including the model's accuracy and loss during training for each epoch
- Generate new samples for each of the given datasets

References & Resources

- [Diffusion Models at Wikipedia](#)
- [Denoising Diffusion Probabilistic Models Paper](#)
- [Stanley Chan, Tutorial on Diffusion Models for Imaging and Vision, 2024.](#)